FIG. 1

(REC)	TID	EVENT TYPE	CATEGORY	SOURCE	APPLICATION	HOST	SEVERITY
(1)	1001	TCPConnectionClose	NETWORK	IO	SYSTEM	VESUVIO	LOW
(2)	1001	CiscoDCDLinkUp	NETWORK	DHCP	ROUTING	ETNA	LOW
(3)	1001	AuditFailure	SECURITY	SOFTWARE	AUTHORIZATION	MAGNA	HIGH
(4)	1001	CoreDump	MEMORY	EXCEPTION	KERNEL 2.4	STROMBOLI	HIGH
(5)	1001	IRQConflict	DEVICE	PCI BUS	SYSTEM	VULCANO	HIGH

FIG. 2

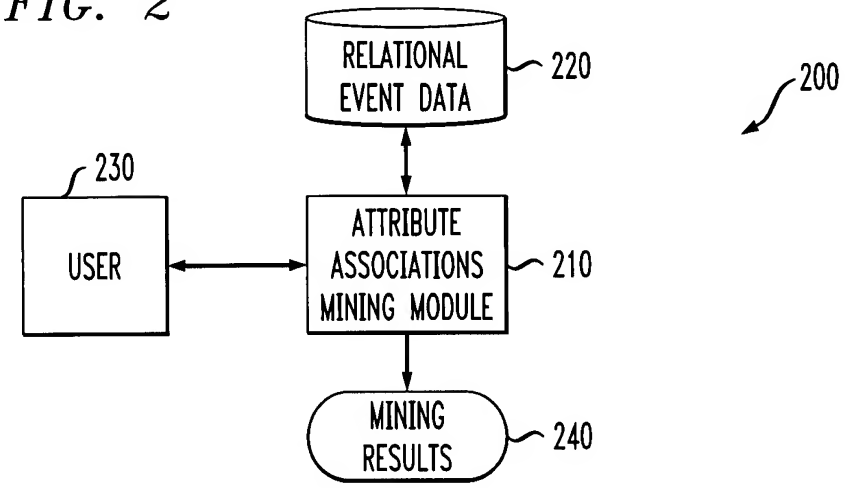

FIG. 3

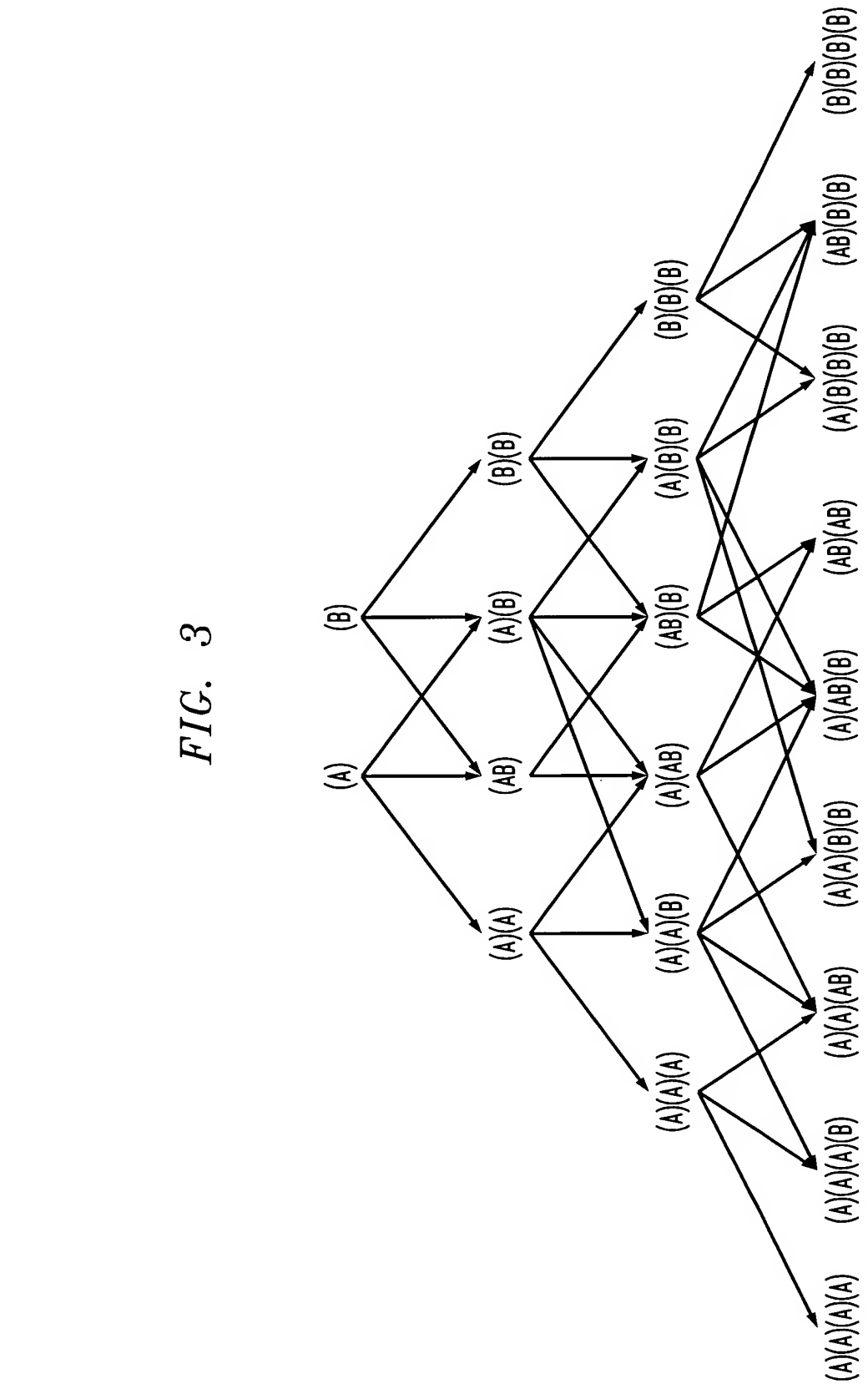

FIG. 4

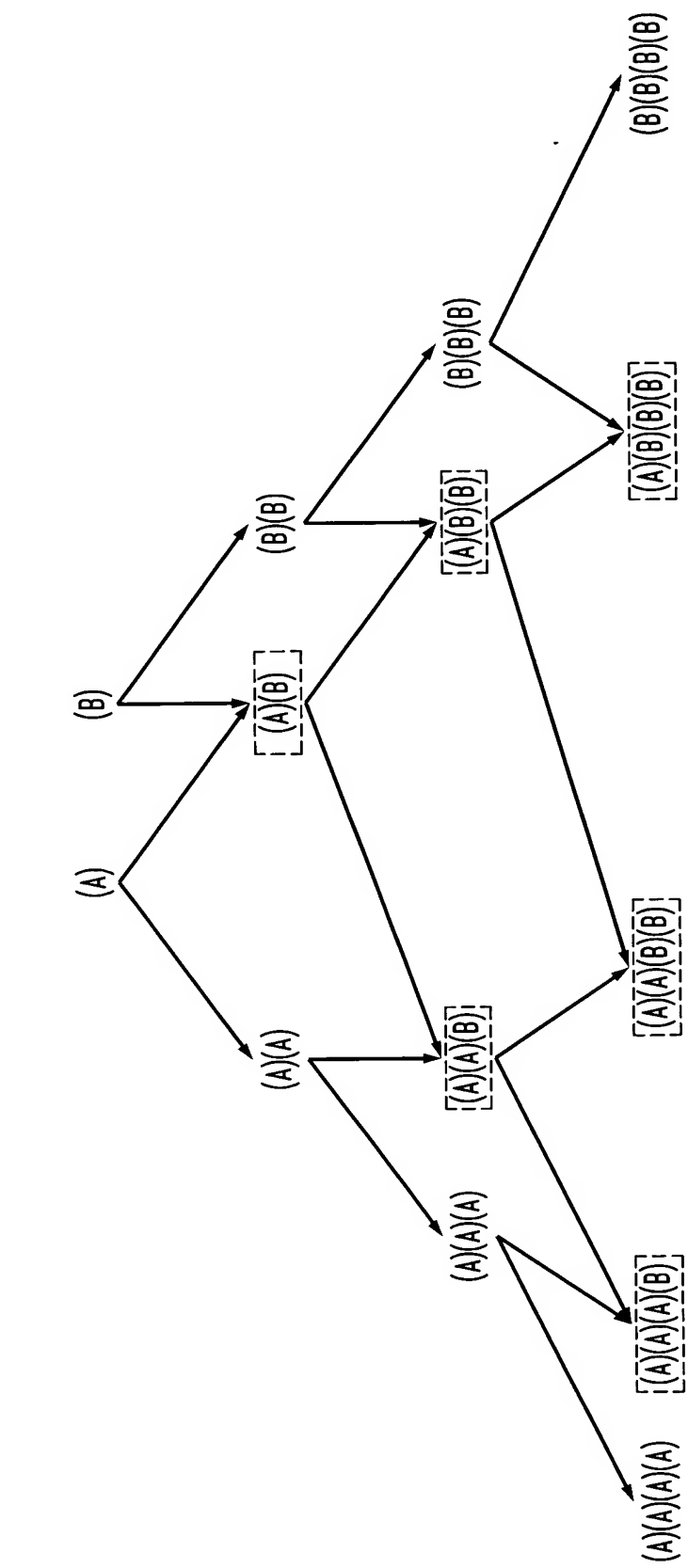

FIG. 5

τ_1	τ_2	
t_{11}:*A*	t_{21}:*B*	t_{22}:*C*

(a) c = (*A*)(*BC*)

τ_1	τ_2
t_{11}:*A*	t_{21}:*B*

(b) c_3 = (*A*)(*B*)

τ_1	τ_2
t_{11}:*A*	t_{21}:*C*

(c) c_2 = (*A*)(*C*)

τ_1	
t_{11}:*B*	t_{12}:*C*

(d) c_1 = (*BC*)

FIG. 6

$c = c_L \bowtie c_{L-1}$	c_L	c_{L-1}	MERGE-JOIN?
(A)(A)(A)	(A)(A)	(A)(A)	YES
(A)(A)(B)	(A)(A)	(A)(B)	YES
(A)(AB)	(A)(A)	(A)(B)	NO
(AB)(B)	(AB)	(A)(B)	YES
(A)(B)(B)	(A)(B)	(A)(B)	YES
(B)(B)(B)	(B)(B)	(B)(B)	YES

FIG. 7

Methodology 1 HIFI(AttributeSet: A, Dataset: D, MinSupport: min_sup)

1: generate frequent patterns for templates on the 1st level;
2: $L \leftarrow 2$;
3: $Template_L \leftarrow$ pair up templates on the 1st level to generate templates on the 2nd level;
4: $Cand_L \leftarrow$ join patterns on the 1st level to generate candidate patterns on the 2nd level;
5: **while** $Cand_L \neq \emptyset$ **do**
6: countSupport(D, $Cand_L$);
7: eliminate candidates whose support are lower than min_sup;
8: $L \leftarrow L + 1$;
9: $Template_L \leftarrow TemplateGen(Template_{L-1}, A)$;
10: CandidateGen($Template_L$);
11: **end while**
12: **return** $\{t.patterns | t \in Template_L\}$;

FIG. 8

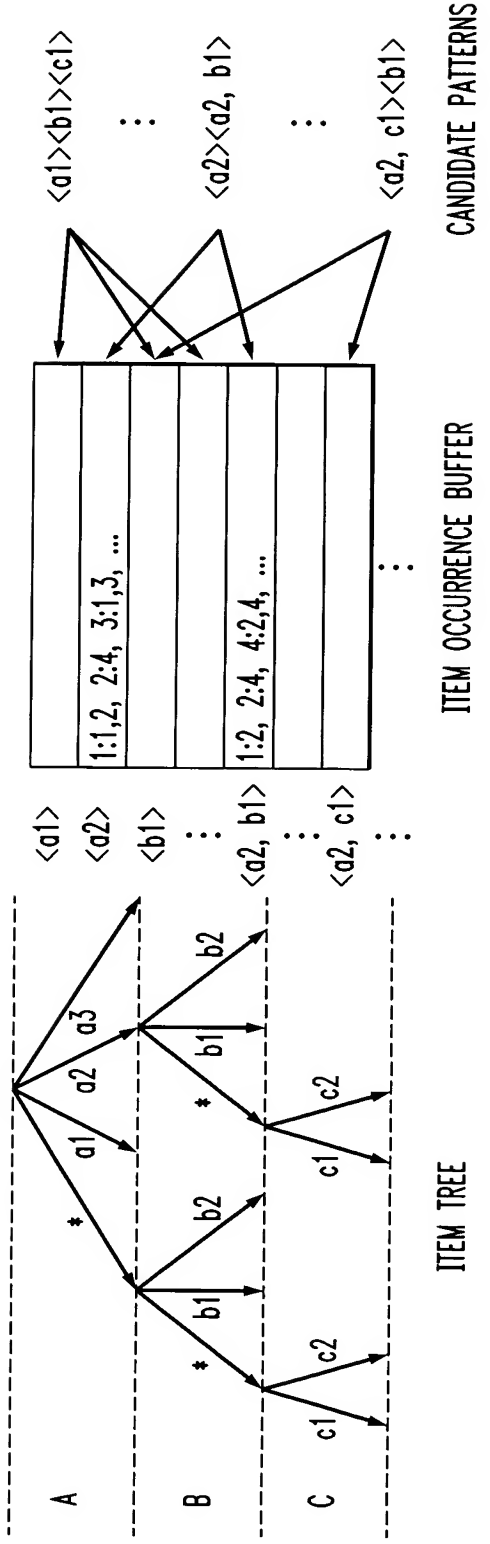

FIG. 9

Methodology 2 countSupport(Dataset: D, PatternSet: *candidates*)

```
1:  for each tuple t ∈ D do
2:    traverse the item tree to find all items supported by t;
3:    for each item supported by t do
4:      record the current TID and RID in item's occurrence buffer;
5:    end for
6:    if any occurrence buffer is full then
7:      for each pattern p ∈ candidates do
8:          scan the occurrence buffer of each item in p (scan is synchronized by TID), and increase the count
            of p if each item of p is supported by a different record (of the same TID);
9:      end for
10:     empty all occurrence buffers;
11:   end if
12: end for
```

FIG. 10

Methodology 3 TemplateGen(SetOfTemplates: *parentTemplates*)

```
1:      Templates ← Ø;
2:  for each p ∈ parentTemplates do
3:      for each child template c of p do
4:        Templates ← Templates ∪ {c} if all of c's parent templates exist and have non-empty pattern set;
5:        prune Templates using user preferences;
6:      end for
7:  end for
8:  return Templates;
```

FIG. 11

Methodology 4 CandidateGen(SetOfTemplates: *Templates*)

```
1:  for each c ∈ Templates do
2:      Let c_i and c_j be the two parents of c that have the fewest number of patterns;
3:      if joining c_L and c_{L-1} is less costly than joining c_i and c_j then
4:          c.patterns ← mergeJoin(c_L, c_{L-1});
5:      else
6:          sort the patterns in c_i and c by their common attributes;
7:          c.patterns ← mergeJoin(c_i, c_j);
8:          sort the patterns in c.patterns;
9:      end if
10:     prune c.patterns if c has value exclusion constraints;
11:     for each patent c_k of c, and c_k does not take part in the join operation do
12:         for each pattern p ∈ c.patterns do
13:             remove p from c.patterns if the sub-pattern of p with regard to c_k does not exists in c_k.patterns;
14:         end for
15:     end for
16: end for
```

FIG. 12